

FIGURE 1

```
// for each pixel in the NxM image
for(x=1; x<N; x+=1)
{
    for(y=1; y<M; y+=1)
    {
        // Consider I(x,y), I(x-1,y), I(x,y-1),... neighborhood of (x,y)
        // If some if I(x-1,y-j) fall out of image boundaries, replace them by 0
        // Do not consider reference pixels (I(x-i,y-j) where i,j=0)
        // Apply (6) to compute e(x,y)
        sum_kI=0;
        for(i=0; i<a; i+=1)
        {
            for (j=0; j<a; j+=1)
            {
                if(I(x-i,y-j)==out of bounds pixel) I(x-i,y-j)=0;
                if(i==0 && j==0) continue;
                sum_kI=sum_kI+kj*I(x-i,y-j);
            }
        }
        sum_kI=sum_kI>>10;
        e(x,y)=I(x,y)-sum_kI;
    }
}
```

FIGURE 2

```

int RTJPEG::CompressRT(BYTE* input, BYTE* output, double ratio, double maxtime,
int bpp=1, bool rgb=false)
{
    int size_orig = Size(input); // original image size in bytes
    // Set max and min quality values
    int q0=5; // 5% quality
    int q1=95; // 95% quality
    // Find respective compression ratios. Higher ratio corresponds to lower quality
    double r0 = size_orig/Compress(input,output,q0, int bpp=1, bool rgb=false);
    // highest ratio
    double r1 = size_orig/Compress(input,output,q1, int bpp=1, bool rgb=false);
    // lowest ratio
    // Return if proposed ratio is smaller than the smallest possible
    if(ratio<r1) return JPEG::Compress(input, output, q1);
    // Return if proposed ratio is larger than the largest possible
    if(ratio>r0) return Compress(input, output, q0);
    // Start timer
    clock_t start, finish;
    start = clock();
    // Start iterative process to estimate the quality value
    double duration=0; double r = ratio;
    while (duration<maxtime)
    {
        // Use linear quality estimate
        int q = q0+(r-r0)*(q1-q0)/(r1-r0);
        // Alternatively, a bi-section estimate can be used in the previous line:
        // int q = (q0+q1)/2;
        // Update estimated corresponding compression ratio value
        r = size_orig/Compress(input,output,q);
        // Update compression and quality boundaries
        if(ratio>r)
        {
            r1 = r; if(q1==q) break; // convergence
            q1 = q;
        }
        else
        {
            r0=r; if(q0==q) break; // convergence
            q0=q;
        }
    }
    // Update timer
    finish = clock(); duration = (double)(finish - start) / CLOCKS_PER_SEC;
    // end of iterative process
    // Compress
    return Compress(input, output, q);
}

```

FIGURE 3

```

//      It is assumed that ApplicationEntityList ael and DICOMObject d variables
//      have been already initialized based on the present network
//      configuration and the data to be sent.

// Find current remote application entity, to which data must be sent
ApplicationEntity ae = ael.GetCurrentEntity();

// Create PDU connection
PDU pdu;

// Start network performance timer
NetworkTimer nt;

// Find data size to be sent
nt.GetDataSize(d);
nt.StartTimer()

// Send data to current application entity
pdu.Send(d,ae);

// Stop the timer, evaluate observed network bandwidth, and store it into ae
nt.EstimateCurrentBandwidth(ae);

```

**FIGURE 4**

```

// It is assumed that ApplicationEntityList ael and DICOMObject d variables
// have been already initialized based on the present network
// configuration and the data to be sent.

// Compress DICOM object to accommodate current network bandwidth
int dsize = d.GetSize();
int ntime = ae.GetDownloadTime(); // maximum allowable download time
int ctime = dsize / ae.GetCurrentBandwidth(); //time we'll spend with current bandwidth
if(ctime > ntime) // low bandwidth, we need compression
{
    if(ae.CanAcceptLossy()) // we can use lossy compression
    {
        // Compress with maximum ratio allowed
        double ratio = min(ctime/ntime, ae.GetMaximumCompressionRatio());
        d.Compress(false,ratio);
    }
    else if(ae.CanAcceptCompressed()) // only lossless, try our best
    {
        d.Compress(true, /*ignored*/1.0);
    }
    else // no JPEG compression allowed, do nothing
    {
    }
}
else // network is fast enough, do not have to compress
{}

// Create PDU connection
PDU pdu;

// Send compressed data to current application entity
pdu.Send(d,ae);

```

**FIGURE 5**

```

class AccurateTimer
{
private :
    int Initialized;
    __int64 Frequency;
    __int64 BeginTime;
public :   AccurateTimer()    // constructor
    {
        // get the frequency of the counter
        Initialized = QueryPerformanceFrequency( (LARGE_INTEGER
*)&Frequency );
    }

    void StartTimer()  // start timing
    {
        if( ! Initialized ) return 0; // error - couldn't get frequency
        // get the starting counter value
        QueryPerformanceCounter( (LARGE_INTEGER *)&BeginTime );
        return
    }

    double EndTimer()  // stop timing and get elapsed time in seconds
    {
        if( ! Initialized ) return 0.0; // error - couldn't get frequency
        // get the ending counter value
        __int64 endtime;
        QueryPerformanceCounter( (LARGE_INTEGER *)&endtime );
        // determine the elapsed counts
        __int64 elapsed = endtime - BeginTime;
        // convert counts to time in seconds and return it
        return (double)elapsed / (double)Frequency;
    }
}

```

**FIGURE 6**

```

bool SoundRecorder::Record(int sf, bool stereo, int max_time=300, int
max_size=1000000, int format=WAV)
{
    DWORD dwReturn;

    // Open a waveform-audio device with a new file for recording.
    if(!OpenMCI()) return;

    MCI_RECORD_PARMS mciRecordParms;
    // Set recording parameters here as fields in mciRecordParms
    mciRecordParms.dwFrom = 0;
    mciRecordParms.dwTo = max_time;
    ....
    mciRecordParms.dwCallback = (DWORD)(this->GetSafeHwnd());

    // Record
    if (dwReturn = mciSendCommand(m_Device.wDeviceID, MCI_RECORD,
MCI_FROM | MCI_NOTIFY, (DWORD)(LPVOID) &mciRecordParms))
    {
        AfxMessageBox(ErrorMCI(dwReturn, "Recording error: ");
        mciSendCommand(m_Device.wDeviceID, MCI_CLOSE, 0, NULL);
        return false;
    }
    return true;
}

```

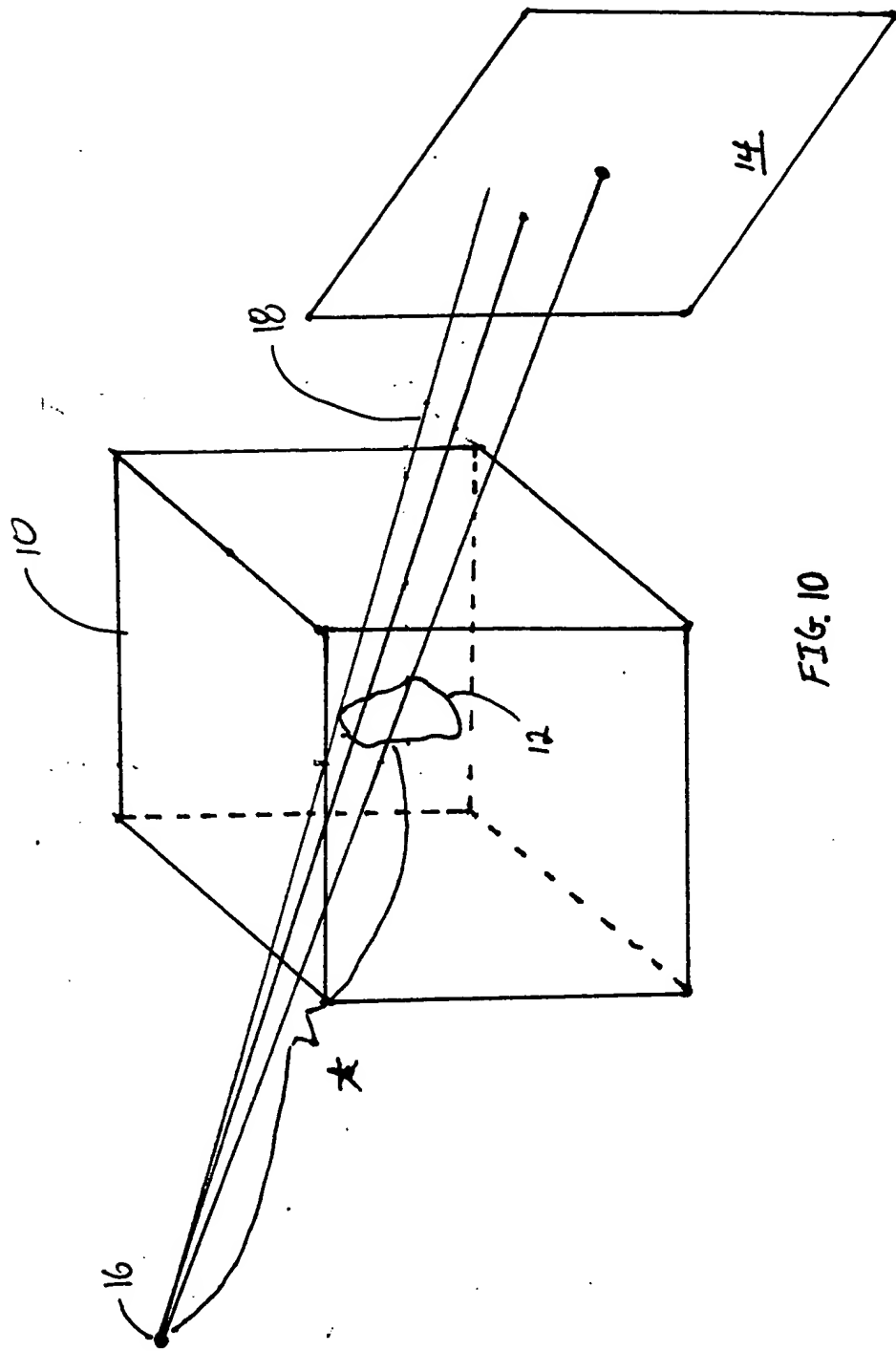
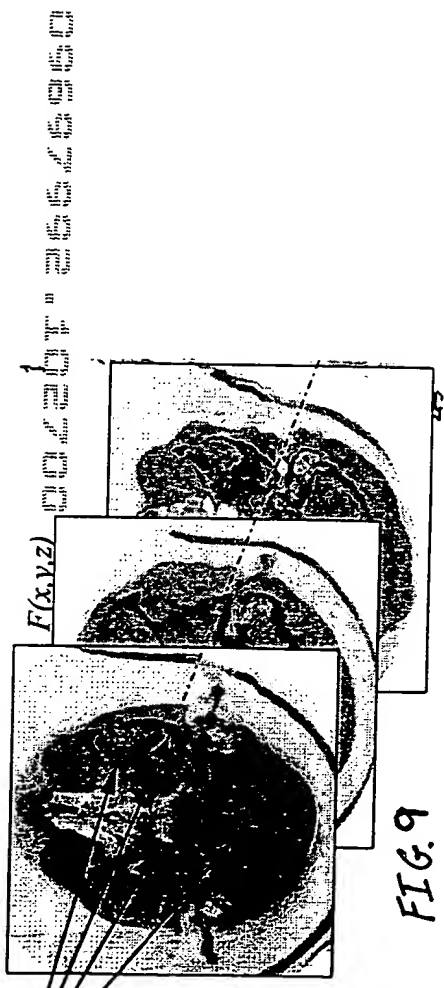
**FIGURE 7**

```

void SoundRecorder::Insert(DICOMObject& dob)
{
    BYTE* sound;
    if(GetFormat() != MP3) // we have to encode the sound
    {
        SoundEncoder se;
        if(se.Encode(GetSound(), GetFormat(), sound)
        {
            return false; // we failed to encode
        }
    }
    // Store sound bytes into odb object
    ....
    return true;
}

```

**FIGURE 8**



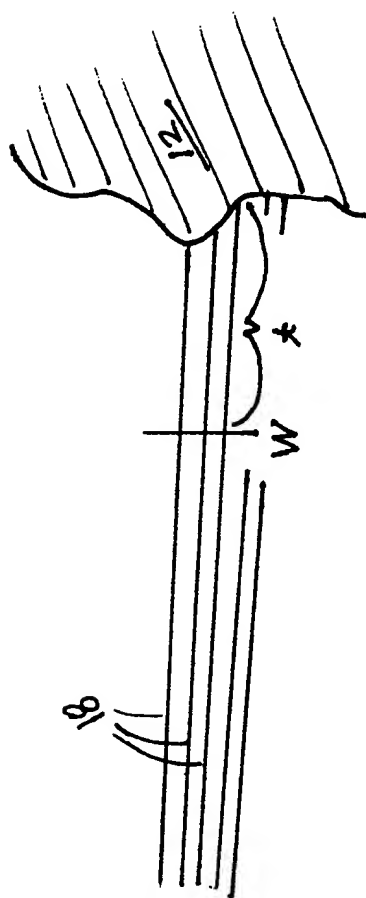


FIG. 11

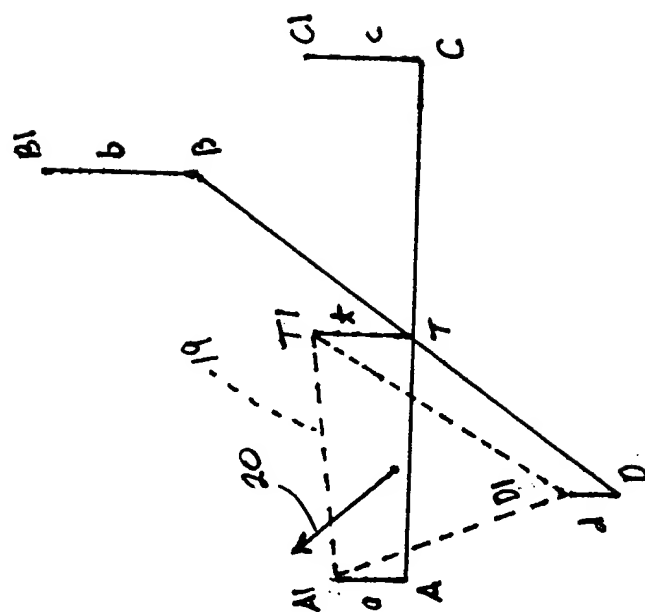


FIG. 12



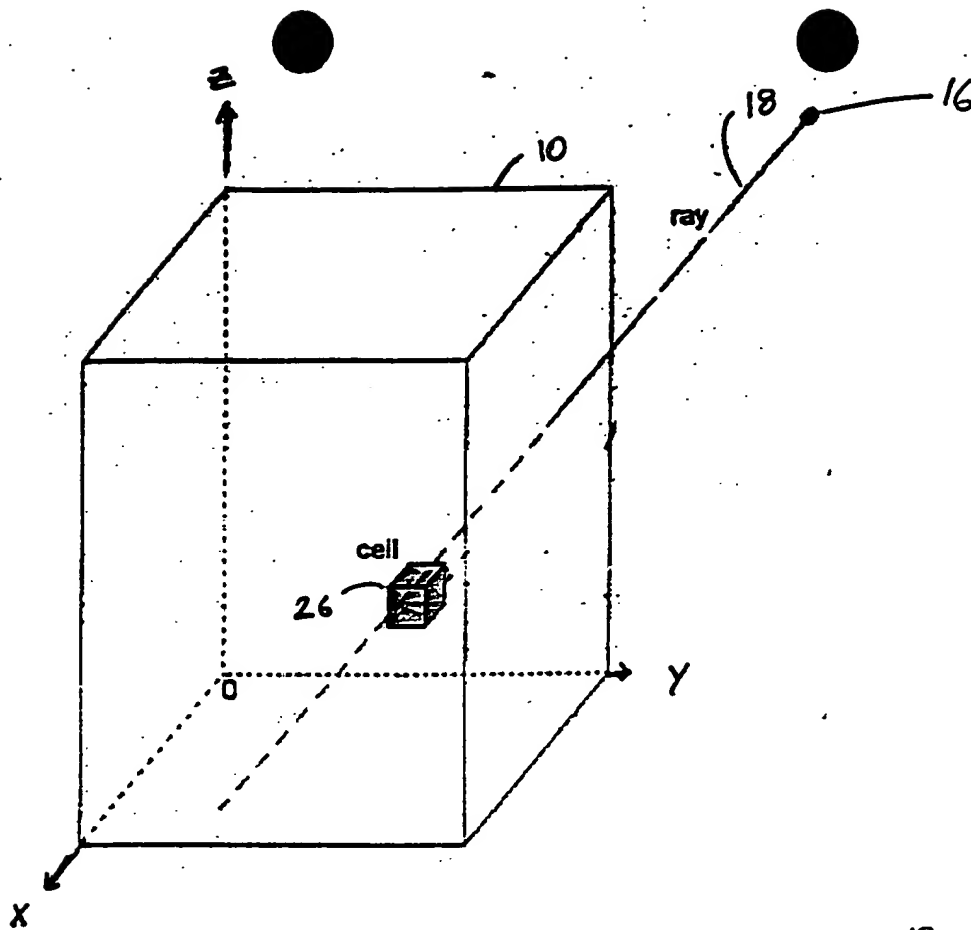


FIG.13 (a)

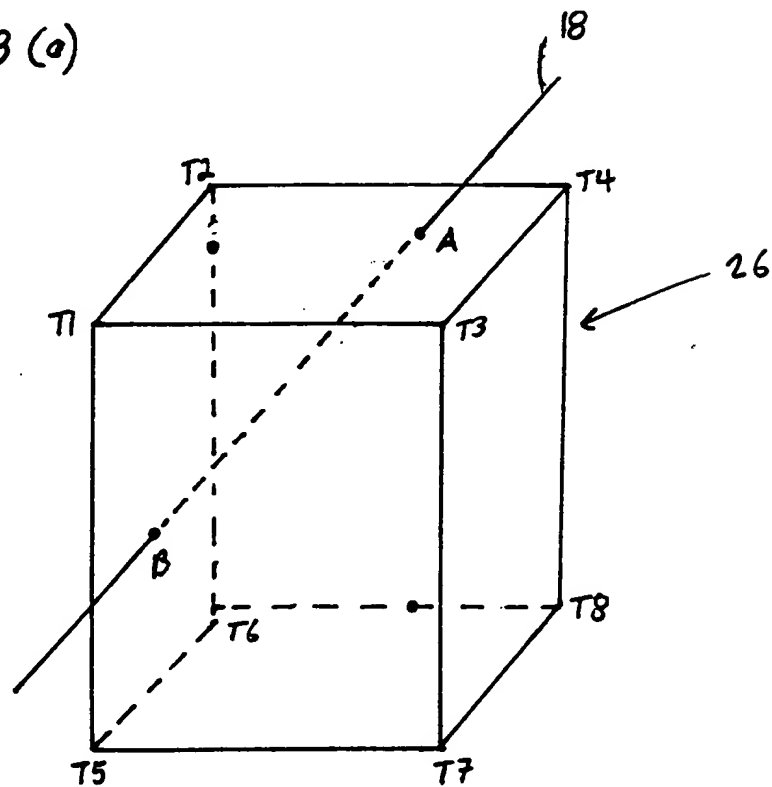


FIG. 13(b)

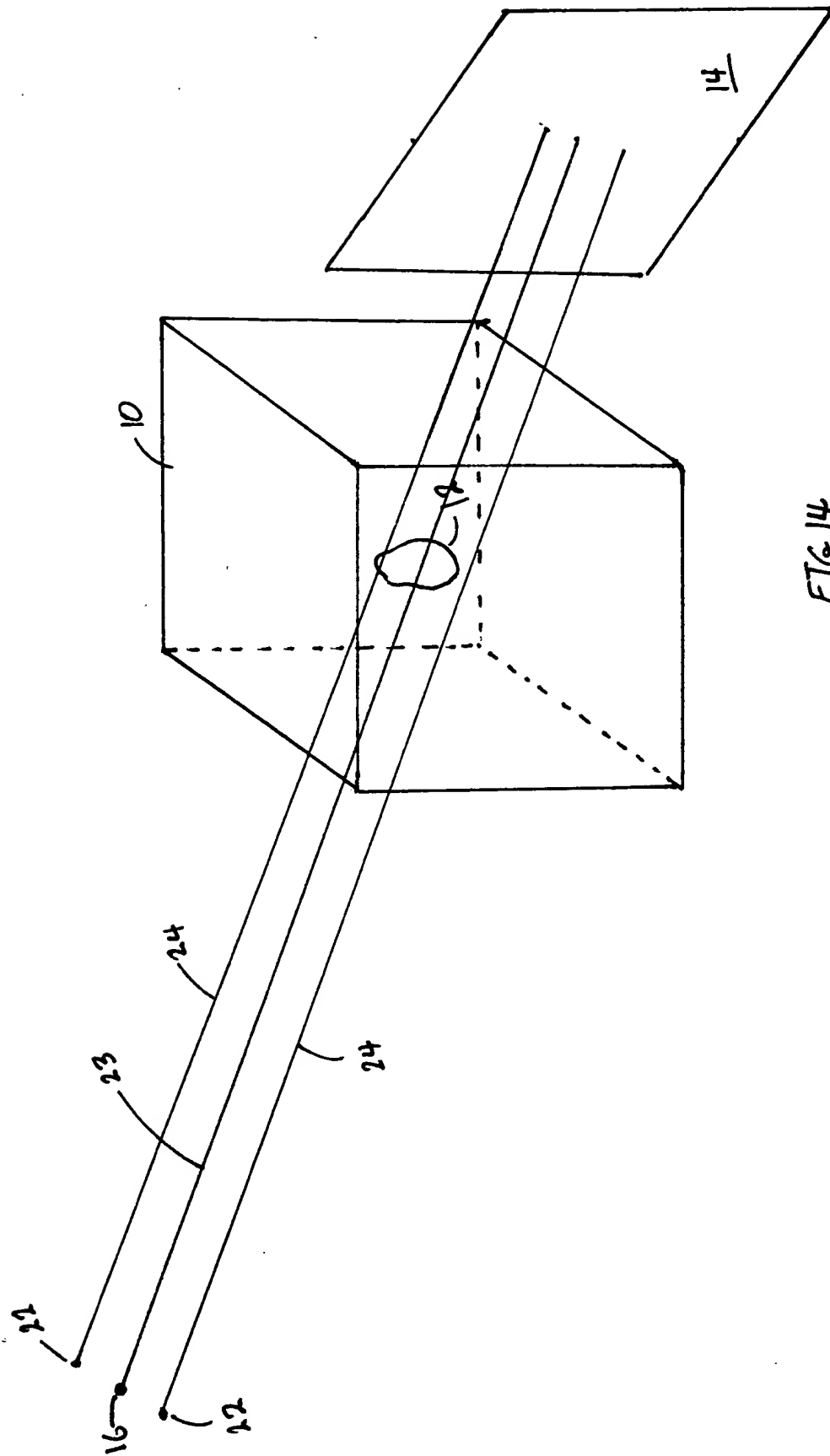


FIG. 14

A diagram illustrating a 3D coordinate system with axes labeled  $x$ ,  $y$ , and  $z$ . A cube, labeled 10, is positioned in the first octant with its origin at the intersection of the axes. A small rectangular cell, labeled 26, is located inside the cube. A dashed line, labeled 24, extends from the origin through the cell and continues outwards, labeled 22. The word "cell" is written above the small rectangular object.

FIG. 15(a)

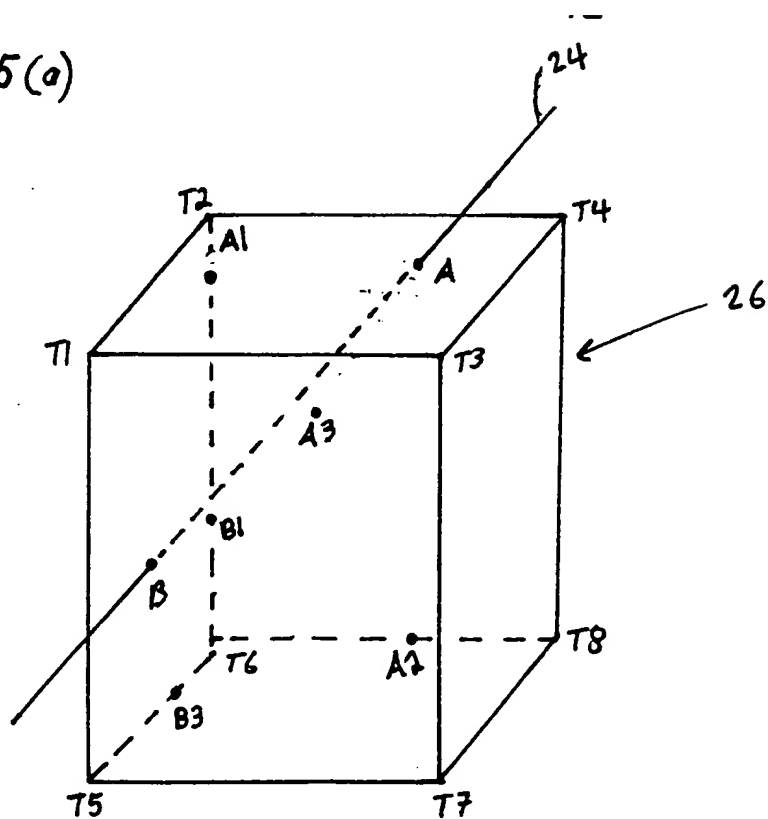


FIG. 15(b)

1 - linear interpolation  
2 - parabolic root interpolation

FIG. 16

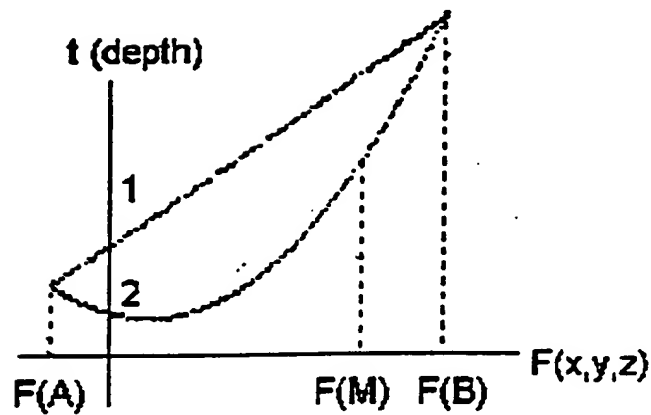


FIG. 17

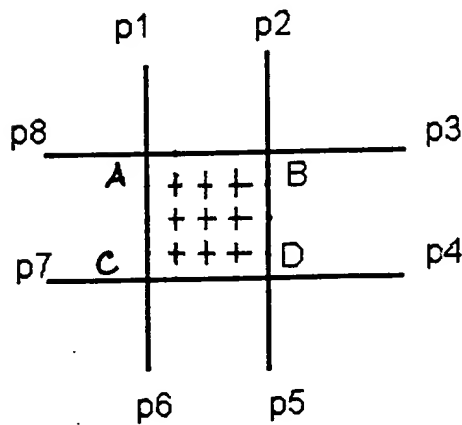
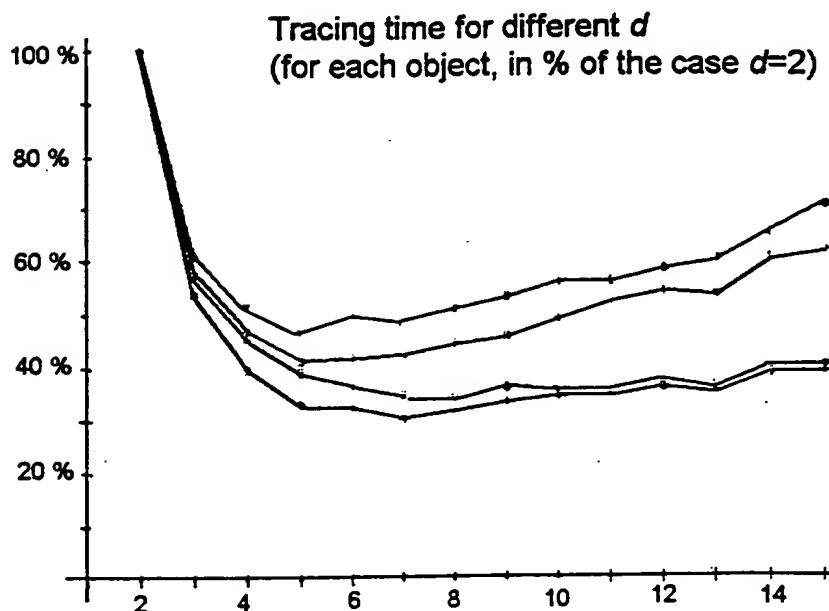


FIG. 18





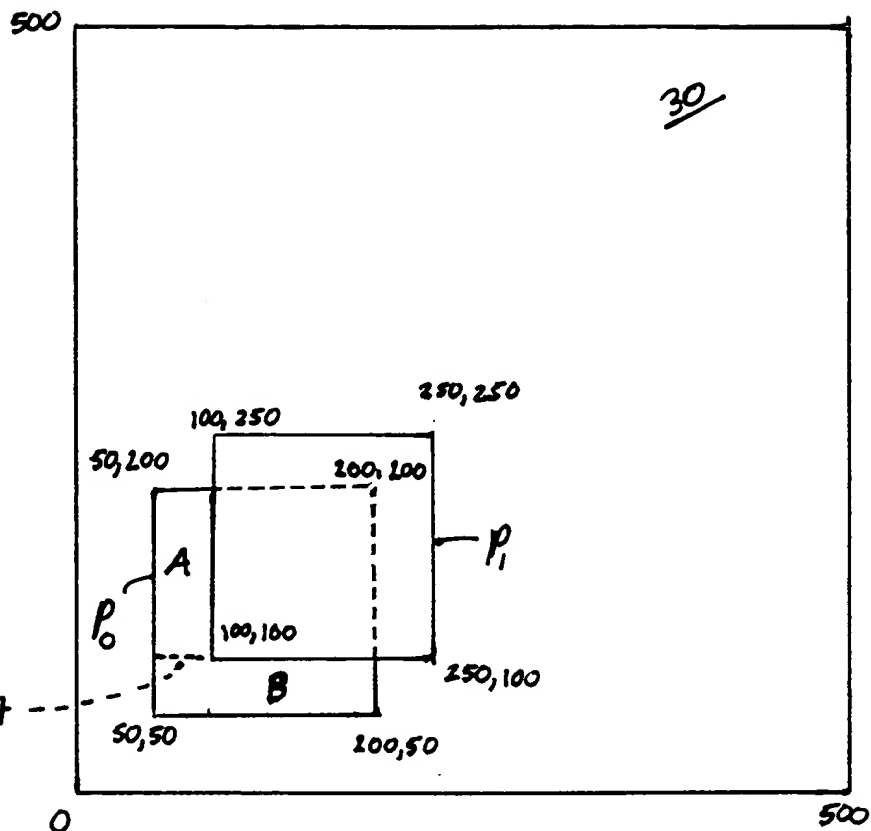


FIG. 21

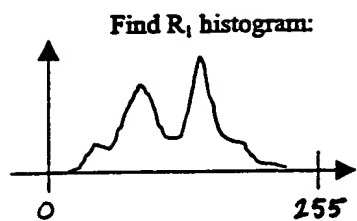


FIG. 23(a)

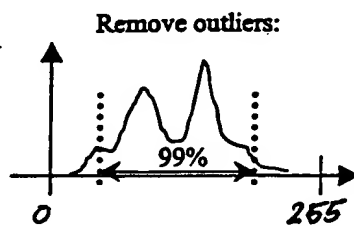


FIG. 23(b)

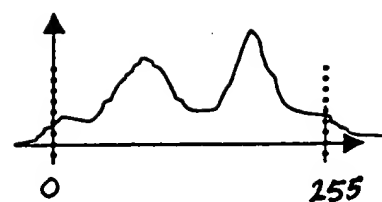


FIG. 23(c)

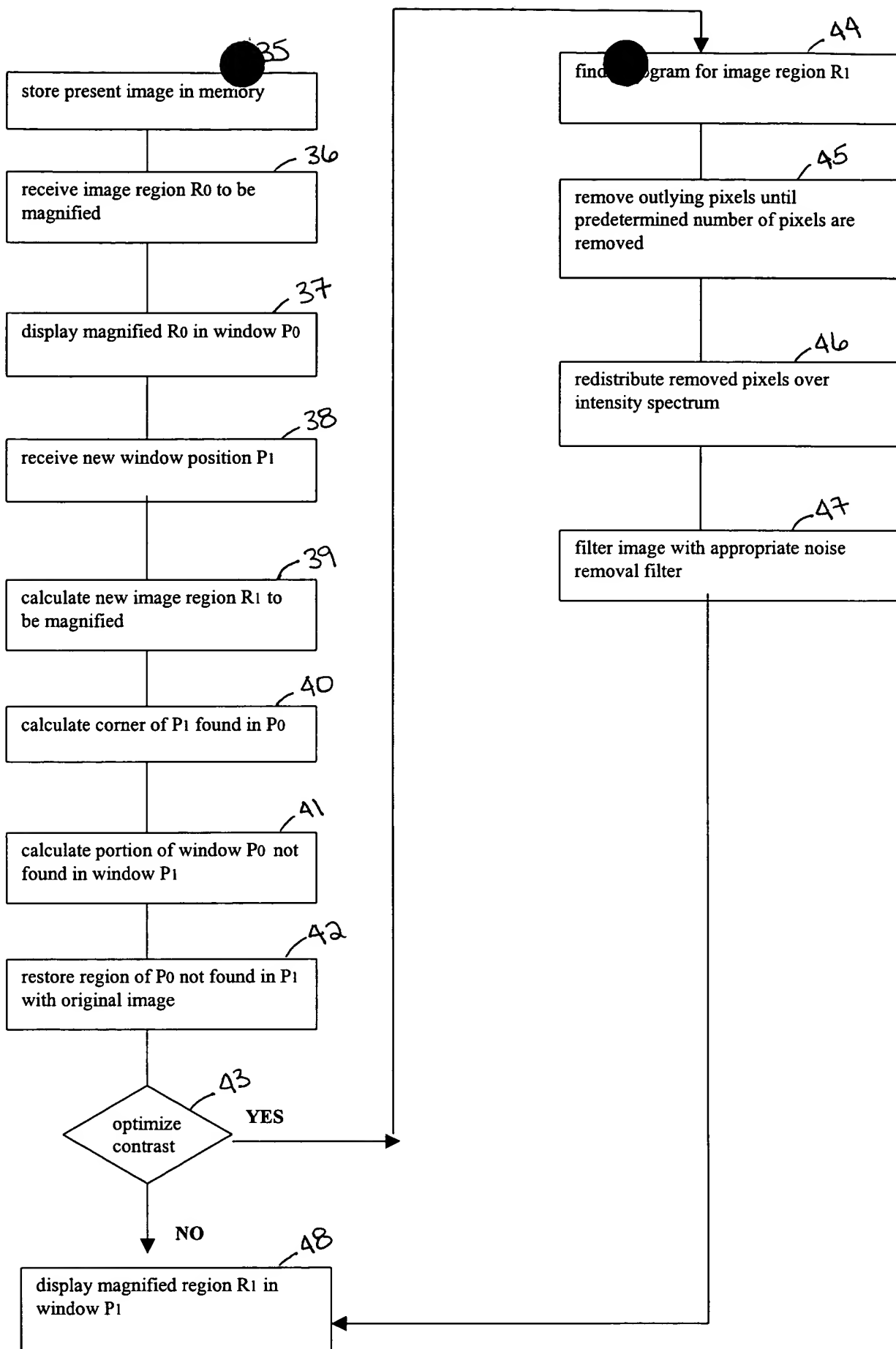


FIG. 22